Printing and Publishing C Programs

Ronald Baecker
University of Toronto

Aaron Marcus
Aaron Marcus and Associates

Program Appearance

Program appearance has changed little since the first high-level languages were developed in the 1960s. With a few exceptions, most notably APL, programs have been expressed and composed almost entirely out of alphanumeric symbols ("ASCII text"). They are typically presented in a single typeface, often without even the use of boldface or italic; in a single point size, with fixed-width characters, fixed wordspacing, and fixed linespacing; and without the benefit of symbols, rules, grids, gray scale, diagrammatic elements, and pictures.

Why is program appearance so impoverished? In part, this is an artifact of the composing and printing technologies of the early days of computing — keypunch, teletype, and line printer. These obsolete technologies have been superseded by the interactive raster display and the laser printer. Programs can now easily be represented using those elements heretofore omitted, such as multiple typefaces; variable weights, slants, point sizes, wordspacing, and linespacing; and rules, gray scale, and pictures.

Yet this should just be the beginning. In our professional and personal lives we employ symbolic systems such as circuits, maps, mathematics, and music that have sophisticated and well-developed notations, appropriate diagrammatic elements, and typesetting and graphic conventions. There are printing and publishing industries dedicated to facilitating effective communication in these disciplines. Why should we not do the same for computer science? Or, in the words of Donald Knuth (1984, p. 97)

> "I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be *works of literature*. Hence, my title: `Literate Programming.'
>
> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do.
>
> The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style... He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that nicely reinforce each other."

Research Claims

In this chapter we seek to demonstrate and explain (as does Oman, 1997) how communication about programs and comprehension of programs can be aided by paying attention to the visual schema embodying the programs and the visual appearance of programs. What appears here is a summary of a previously published book-length treatment of the same material, *Human Factors and Typography for More Readable Programs* (Baecker and Marcus, 1990).

We show that the presentation of program source text matters, and demonstrate, by existence proof, that it is possible to produce significantly better program presentation than that to which we are accustomed. Effective program presentation portrays program structure and helps us deal with its complexity, making good programs more understandable and bad programs more obvious.

Effective program presentation, visually enhanced source code, and improved legibility and readability can be achieved through the systematic application of graphic design principles (Marcus, 1990; Marcus, 1995). The design for the appearance of a language can and should be documented in a *graphic design manual* for that language. Our prototype graphic design manual for C appears in Baecker and Marcus (1990).

The fact that principles are applied to achieve a design does not imply that the result is uniquely determined. There are many plausible designs that may be derived from the principles. They must be evaluated by generating page after page of visual examples and by thoughtful scrutiny and testing of the strengths and weaknesses of each. Examples of these *systematic design variations* for the appearance of C appear in Baecker and Marcus (1990), and represent the first published “catalog” of possibilities for the appearance of program source text.

Effective program appearance may be automated. Most of the examples included in Baecker and Marcus (1990) have been produced by a *visual compiler* for the C language, which takes unmodified C source text as its input, and produces as its output high-quality typeset presentations on a laser printer. The SEE visual compiler is also heavily parameterized to allow the customization of text displays to suit individual taste.

The concepts developed in this research may be applied to other programming languages. With languages similar to C, like Pascal, the principles and specifications are easily extendible; with those more distant, such as Prolog, significant new work will be required.

Enhanced program presentation produces listings that facilitate the reading, comprehension, and effective use of computer programs. Baecker and Marcus (1990) and Oman and Cook (1990a,b) present both theory and experiments which lend credence to the following assertion: *Making the interface to a program's source code and documentation intelligible, communicative, and attractive will ultimately lead to significant productivity gains and cost savings.*

Our approach can be considered part of a broader effort to enhance the art of program writing, documentation, and illustration, Knuth’s “literate programming” (see also Ramsey, 1994). We have considered the entire context in which code is presented and used, a context which includes the supporting texts and notations that make a program a living piece of written communication. We seek thereby to enhance programmers' abilities to construct, refine, store, retrieve, scan, read, and manipulate the texts and documents required to do their jobs.

To summarize, our goal is for computer science to learn and apply the lesson that the discipline of graphic design teaches so vividly: *Visual form matters. Effective representation and presentation aids thought, the management of complexity, problem solving, and articulate expression.*

An Example of a C Program Designed and Typeset

We present as Figures 1 and 2 a short, self-contained illustration of fairly typical C code that accepts a phone number expressed as a sequence of digits and prints out a list of equivalents with all digits between two and nine replaced by corresponding letters from a phone dial, with the constraint that the words produced must each contain at least one vowel. Thus, given "765" as its input, the program produces the following as its output: poj, pok, pol, roj, rok, rol, soj, sok, and sol. Figures 1 and 2 show SEE's output, altered only in the addition of details in headers and footnotes that were not handled automatically by SEE.

Design Method

To develop this new software visualization technique, we followed traditional analysis and design procedures as follows:

Visible Language Taxonomy

We first produced a visible language taxonomy for computer-based documents and publications (see also Gerstner, 1978; Ruder, 1973; Chaparos, 1981). This organization was intended to be a checklist to guide approaches to enhancing source code presentation.

C Taxonomy

We simultaneously developed a taxonomy of C constructs, a systematic enumeration and classification of logical components of the language (AT&T, 1985; Kernighan and Ritchie, 1978; Harbison and Steele, 1984). This was intended to be a companion checklist for ensuring completeness in the representation of C source text. We also added notions not in the formal language description, for example dealing with various categories of comments.

Review of Current Practice

Next, we collected, organized, and reviewed typical mappings from C constructs to visible language constructs, examples abstracted from real C programs prepared by typical experienced C programmers. These "folk designs" often embody valuable design insights from non-designers, as opposed to the "professional designs" such as the conventions proposed in this book.

Design Principles

We then developed a systematic set of principles that govern the design of mappings from C constructs to visible language constructs. These principles guide detailed visual research into the effective presentation of C source code.

A New Design

We applied these principles and informally reviewed the merits of interim designs to develop a new set of conventions for the presentation of C programs. The design specifications have been illustrated by applying them to the concrete example presented above.

Design Refinement

To facilitate our systematic approach to the design of program presentation, we constructed SEE, a visual C compiler, a program that maps an arbitrary C program into an effective typeset representation of that program. We produced numerous examples using this automated tool, which has in turn enabled us to modify and improve the graphic design of program appearance.

Design Testing

The design was also tested informally and more formally in an experiment in an attempt to substantiate and quantify its value.

Design Formalization

The final specifications (that is, a precise, complete mapping of all elements of C to selected appearance characteristics) were then embodied in a graphic design manual for the appearance of C programs.

Iterative Design

We did not proceed through these steps in a linear fashion. Analysis of examples produced with the help of the visual compiler, feedback from members of the project team and other interested parties, and results from experiments were all used in an ongoing process of *iterative design* refinement and improvement.

Program Publishing

Finally, we shifted our viewpoint away from code appearance and considered the larger issue of the function, structure, contents, and form of the *program book*, the embodiment of the concept of the program as a publication. (See also Oman and Cook, 1990a,b, for a similar development of this concept.) Although we did not automate its production, we developed and included in Baecker and Marcus (1990) a mock-up of a prototype of a program book. We shall now, following Figures 1 and 2, turn our attention to this aspect of our work.

Figure 1. Page 1 of the designed and typeset program. (Baecker and Marcus, 1990, p. 9).

We explain our design's salient features with reference numbers, e.g., ***(1)*** *and* ***(17)****, that refer to the small numbers in circles appearing in the right margin of the corresponding program pages. The notes are organized in terms of a taxonomy of C constructs introduced in Baecker and Marcus (1990).*

The Presentation of Program Structure

*The program is output on loose-leaf 8.5" X 11" pages, each of which is separated into four regions, a header (****1****), a footnote area (****17****), a main text column for the code and most of the comments (****3****, right), and a marginalia comment column (****3****, left).*

*Each file appears as a separate "chapter" with the filename shown as a very large, bold title (****2****).*

Extra white space is used to provide adequate separation between the ***prologue*** *comments (see below) and the code (****5****), between function definitions and sequences of declarations (****10****), between individual function definitions, and between the header and the body of a function definition (****14****).*

*Cross-references relating uses of global variables to the location of their definitions are included as footnotes to the source text (****17****).*

The Spatial Composition of Comments

*Each file may include, at or near its beginning, a prologue comment describing the module's purpose (****4****).*

The prologue is displayed in a serif font over a light gray tone. Type size and gray value have been selected to insure legibility of two generations of photocopies. There is a margin around the text ample to ensure readability.

Comments that are located on the same lines as source code, which we call *marginalia* comments, are displayed in a small-sized serif font in the marginalia column (**9**, left). These items are intended to be short, single-line phrases.

The Presentation of Function Definitions

The introductory text of a function definition — the function name — is shown as a "headline," in a large sans-serif type (**11**). A heavy rule appears under the introductory text of a function definition (**12**). A light rule appears under the declaration of the formal parameters (**13**).

The Presentation of Declarations

Identifiers being declared are aligned to a single implied vertical line located at an appropriate horizontal tab position, 16 picas into the main code column (**7**).

Initializers are displayed at reasonable tab positions, with the programmer's carriage returns being respected as requests for "new lines" (**8**).

The Presentation of Preprocessor Commands

The "**#**" signifying a preprocessor command is exdented to enhance its distinguishability from ordinary C source text (**6**).

Within macro definitions, macros and their values are presented at appropriate horizontal tab positions, 8 and 16 picas into the main code column (**6**).

The Visual Parsing of Statements

Systematic indentation and placement of key words based on the syntax of the program is employed (**15**).

Since curly braces are redundant with systematic indentation, they are removed in this example (**15**). Whether this happens or not is under control of the user.

In conventional program listings, it is impossible to tell without turning the page where a particular control construct (in this case, the *for*) continues on the following page. Our solution is an ellipsis, in line with the *for*, signifying that the first statement on the next page is at the same nesting level as the *for* (**16**).

Figure 2. Page 2 of the designed and typeset program (Baecker and Marcus, 1990, p. 11).

The Spatial Composition of Comments (continued)

Comments that are **external** *to function definitions are displayed in a serif font laid over a light gray tone (***26***).*

Comments that are **internal** *to function definitions are also displayed in a serif font laid over a light gray tone, appropriately indented to match the current statement's nesting (***28***).*

The Presentation of Function Definitions (continued)

*The function type specifier, indicating the type of the value returned by the function, if any, appears on a line by itself above the function name (***27***).*

The Visual Parsing of Statements (continued)

*When nested statements cross a page boundary, the "nesting context" is displayed in the second column of the header (***19***), just above the code.*

The Visual Parsing of Expressions

*Parentheses and brackets are emboldened to call attention to grouped items. Nested parentheses are varied in size to aid parsing (***25***).*

Unary operators such as **++** *and the unary ** are raised (turned into superscripts) to make them easier to distinguish from binary operators (***30***).*

*The wordspacing between operators within an expression is varied to aid the reader. Operands are displayed closer to operators of high precedence than to operators of low precedence (***29***).*

Typographic Encodings of Token Attributes

Most tokens are shown in a regular sans-serif font; reserved words are shown in italic sans-serif type (**20**).

Since the global variable in C is a fundamental mechanism through which functions communicate indirectly, and thus also a major source of programming errors, we call attention to most uses of globals (but not function names, invocations, or manifest constants) by highlighting them in boldface (**20**).

Macro names, which by C convention are all upper case, are shown with the first letter normal size and the remainder of the word in "small caps" (**22**). String constants are shown in a small-sized fixed-width serif font (**21**).

The Typography of Program Punctuation

In this example the ";" appears in 10 point regular Helvetica type, and thus uses the same typographic parameters as does much of the program code. On the other hand, in order to enhance legibility and readability, the "," has been enlarged to 14 points, the "*" has been enlarged to 12 points, and the "!" has been set in boldface (**24**). Slight repositioning has also been carried out on individual punctuation marks.

The letterspacing between individual characters of multi-character operators such as the "!=", the "<=", and the "++" has been adjusted, to make the symbols more legible as a unit (**23**).

Symbol substitutions are employed where they can reduce the possibilities for error and thereby enhance readability. Since C's two uses of *while* can be confused under some circumstances, we add an upwards-pointing arrow to the *while* that is part of a *do...while* (**25**).

The Presentation of Program Environment

The header describes the context of the source code that appears on the page, including the location of the file from which the listing was made, the last time the file was updated, the page number within the listing, the time the listing was made, and, in the second column, the function name of the code that first appears at the top of the page (**18**).

Programs as Publications

Programs are publications, a form of literature. English prose can range in scope from a note scribbled on a pad to a historical treatise appearing in multiple volumes and representing a lifetime of work. Similarly, programs range from a two-line *shell* script created whenever needed to an edition of the collected works of a research group, for example, the UNIX operating system. The line printer listing, which represents the output of conventional program publishing technology, is woefully inadequate for documenting an encyclopedic collection of code such as the UNIX system, or even for such lesser program treatises as compilers, graphics subroutine packages, and database management systems.

The problem is that computer program source text consisting of code and accompanying comments does not itself have sufficient communicative depth. A program is a large document, an information narrative in which the components should be arranged in a logical, easy-to-find, easy-to-read, and easy-to-remember sequence. The reader should be able quickly to find a table of contents to the document, to determine its parts, to identify desired sections, and to find their locations quickly. Within the program source text, the overall structure and appearance of the page should furnish clues regarding the nature of the contents. The page headers and footnotes should also serve to reinforce the structure and sequencing of the document.

Other document elements that aid and orient the reader are needed. For example, a published program requires an *abstract*, a summary of the function, significance, and capabilities of the program. It should have several kinds of *overview* and *index* pages, all providing the reader with summaries of other facets of the program's function, structure, and processing. The program should also be augmented by a variety of kinds of documentation, some for its users and some for its programmers and maintainers.

Program Publications

More formally, we define a *program publication* as a document (either paper or electronic) consisting of program *text* and *metatext*. The text is the program proper, its source code and comments. The metatext is the body of supporting texts and illustrations that augment, describe, clarify, and explain the text.

A more elaborate concept of a program publication can be formulated in terms of the concepts of *primary text*, s*econdary text*, and *tertiary text*:

• Primary text includes what typically appears in a program listing: the program's source code and comments.

• Secondary text is metatext that augments the primary text directly on the program listing pages. Examples of secondary text are headers, footnotes, and annotations. These typically are *metadata* describing the context in which the program is used, and short *commentaries* (some mechanically produced) pointing out salient features of the program.

• Tertiary text is metatext appearing on pages that supplement the program pages. Tertiary text is the source of additional information about the program, how it was built, and how it is to be used. Examples of tertiary text include the overview and index pages described above, as well as the longer descriptions and explanations of the program that typically are called documentation.

Program Views

More specifically, we have invented and systematized a set of *program view*s, presentations of program source text or representations computed from program source text or program execution that enrich the documentation of the source text and thereby aid the design, construction, debugging, maintenance, and understanding of the program's function, structure, and method of processing.

Each new method encapsulates essential aspects of the program's function, structure, or behavior and provides an answer to a question about the program that could reasonably be asked by an experienced programmer. Thus each technique adds richness that aids in communicating the meaning of a program to its readers and users and can help programmers in carrying out required tasks.

Baecker and Marcus (1990) proposes and gives examples of twelve kinds of program views:

• *Program source text* consists of source code, i.e., text expressed within the programming language, and comments, i.e., text written in English or some other natural language that is embedded within source code and for the most part ignored by processors of programs in that programming language.

• *Program page metadata* appear in the headers and footnotes of program pages and clarify the context in which code on a particular page is to be understood.

• *User documentation* is English prose created to help the user understand the purpose, functionality, and use of the program.

• *Program documentation* is English prose created to help the programmer understand the design and construction of the program.

• *Program introductions* are intended to help a reader of a program publication become oriented with respect to the entire series of documents. In books, such introductions are known as "front matter."

• *Program overviews* are concise textual or diagrammatic descriptions or summaries of the program's function, structure, or processing.

• *Program indices* are sorted lists of program elements organized in such a way as to facilitate reference and access to groups of "related" program elements.

• *Structured program excerpts* are presentations of fragments of program source text within the context of "surrounding" or "nearby" text, in which some text has been omitted (elided) in order that the most relevant information be displayed on the page in relative proximity to the program source fragments.

• *Program change descriptions* are presentations of program source text or fragments thereof showing how the program has changed through one or more recent revisions. This component is extremely important for large programs created by teams of people that require version management and control.

• *Program annotations* are superimpositions on pages of program source text of various metatext explaining and clarifying the program's authorship, history, function, structure, processing, problems, or other useful information. Given appropriate technology, these could include commentaries (possibly hand-written) by previous readers and/or maintainers of the software.

• *Program illustrations* are graphic presentations, e.g., charts or diagrams, that explain or clarify aspects of the program's function, structure, or processing.

• *Program animations* are dynamic program illustrations, i.e., "movies" depicting the program in execution.

A Prototype Program Publication

To illustrate these concepts, we include as Figures 3, 4, and 5 miniature pages from our program publication prototype. The example is based on the famous Eliza program devised by Joe Weizenbaum (1969). The new implementation was written in C by Henry Spencer and modified by Alan J Rosenthal. The SEE program visualizer was used to produce listings of the source text of Eliza, which were modified only in terms of better pagination and the adding of nesting information and footnotes, features not handled automatically by SEE.

These listings were then been combined with metadata, commentaries, indices, overviews, user documentation, system documentation, and other program views to form the program book. Most of the program views began with relevant data about the sample program or its execution being produced automatically using existing or new UNIX tools and utilities. These were then processed using other tools and utilities into an appropriate TROFF or Postscript form suitable for printing. It would not be terribly difficult to automate these processes.

The examples we chose for the program book were meant to be illustrative rather than exhaustive. Numerous other kinds of documentation, such as lists of requirements and system specifications, could also have been included. The program book is organized as follows:

• The book begins with introductions — tertiary text which may include, for example, a cover page (Baecker and Marcus, 1990, p. 147, included as Fig. 3a), title page (p. 149, included as 3b), colophon, abstract and program history page (p. 151, included as 3c), authors and personalities page (p. 152, included as 3d), and table of contents page (2 of 3 pages, pp. 153-154, included as 4a,b).
• Chapter 1 is tertiary text that comprises the user documentation: a tutorial guide (first page, p. 158, included as 4c), command summary, and user manual.
• Chapter 2 contains program overviews, i.e., tertiary text such as a program map (p. 170, included as 4d), call hierarchy, function call history, and execution profile.
• Chapters 3 through 10 is the primary text — the program code and comments. Each program file is a separate chapter. Each program page (pp. 176-7 are included as 5a,b) has useful secondary text included in its headers and footnotes.
• Chapter 11 contains the programmer documentation, i.e., tertiary text such a the installation guide and the maintenance guide.
• Chapter 12 contains indices, e.g., tertiary text in the form of a cross-reference indec, caller index, and callee index (one page, p. 217, is included as 5c).
The last miniature is the back cover page (p. 220, included as 5d).

Figure 3a-d. 4 miniatures of pages from a C program book. a) is the book's cover page, including the title, list of authors, and an illustration. b) is the book's title page, stating the title, authors, and publisher. c) is the abstract and program history page, providing both a summary of what the program does and a capsule history of its development. d) is the authors and personalities page, designed to acquaint readers with key individuals in the development and maintenance of the program.

Figure 4a-d. 4 miniatures of pages from a C program book. a) is one of the table of contents pages, describing the overall structure of the book including primary, secondary, and tertiary text. b) is one of 2 table of contents pages describing the program text, in which each file appears as a separate chapter. c) is one of the pages of user documentation, the beginning of a tutorial guide. d) is one of a set of program overviews, a program map consisting of condensations of all program source text pages. At normal size, one can see features of the program source text which are useful for orienting oneself within the text.

Figure 5a-d. 4 miniatures of pages from a C program book. a,b) are the first two pages of the program source text, typeset and printed according to the design guide described in this paper. c) is one of a set of indices into the source text, this one listing each called procedure and the name and location of every caller of that procedure. d) is the back cover of the program book, giving high-level information about the program which would be useful to those considering its acquisition.

Although our example is shown in paper, a program publication would clearly be more useful if it were an *electronic book*. Information should be interactively computed, generated, and presented (and perhaps then printed) upon demand, based on requests and specifications from the user (Small, 1989). This will aid programmers in obtaining maximal insight from concurrently displayed representations of a program.

Significance and Accomplishments

In our research (Baecker and Marcus, 1990), we applied the tools of modern computer graphics technology and the visible language skills of graphic design, guided by the metaphors and precedents of literature, printing, and publishing, to suggest and demonstrate in prototype form that enduring programs should and can be made perceptually and cognitively more accessible and more usable. (See also Oman, 1997, for a current review of approaches to achieving this goal.)

In carrying out this work, our goal was to make computer program source text more valuable for the programmer, that is:

• more *legible*, meaning that individual symbols comprising the program should be clearly discriminable and recognizable.
• more *readable* and *comprehensible*, meaning that we should be able to read a text with greater speed and deeper understanding.
• more *vivid*, meaning that the text should be able to stimulate our thoughts and our imagination.
• more *appealing*, so that we enjoy and appreciate the experience.
• more *memorable*, so that we are better able to recognize and recall what we have read.
• more *useful*, in terms of such common programmer tasks as scanning, navigating, manipulating, posing hypotheses and answering questions, debugging, and maintaining the program.

At first glance, it may have seemed that we are talking only about work on “prettyprinting”. (see History Chapter by Baecker) Our work, however, goes significantly beyond conventional approaches to prettyprinting in several ways:

• The availability of rich typographic and pictorial representations with many more degrees of freedom changes the nature of the problem to one that is *qualitatively different* from that of prettyprinting on a line printer.

• We have identified basic graphic design principles for program visualization and developed a framework for applying them to programming languages.
• We have systematically carried out graphic design experimental variations in order to arrive at a carefully considered set of design guidelines and recommended conventions for program appearance.
• We have formalized these guidelines and specifications in a graphic design manual for C program appearance.
• We have developed a flexible experimental tool, the SEE *visual compiler*, which automates the production of enhanced C source text. SEE is highly parametric, thus allowing easy experimentation in trying out novel variations, and also suiting the great variety of style preferences that characterizes the community of programmers.
• Finally, we have enlarged the scope of the study of program printing from the narrow issue of formatting source code and comments to a far broader concern with programs as technical publications. This includes an investigation of methods for designing, typesetting, printing, and publishing integrated bodies of program text and metatext, and also the invention of new displays of essential program structure, called *program views*, designed to help programmers master software complexity. In the spirit of Knuth (1984), *Human Factors and Typography for More Readable Programs* helps to establish programming as a literary form deserving a mature graphic appearance.

Acknowledgments

Many people, all acknowledged in detail on pp. xiii-xiv of Baecker and Marcus (1990), contributed to this research. We are especially grateful to Michael Arent, Ilona Posner, Hugh Redelmeier, and Alan J Rosenthal for their competent and dedicated work on this project, and to the US Defense Department’s Advanced Research Projects Agency and to Canada’s Natural Sciences and Engineering Research Council for financial support.

References

AT&T Bell Laboratories (1985). *The C Programmer's Handbook*. Prentice-Hall.

Baecker, R.M. and Marcus, A. (1990). *Human Factors and Typography for More Readable Programs*. ACM Press, Addison-Wesley.

Chaparos, A. (1981). *Notes for a Federal Design Manual*. Washington, D.C.: Chaparos Productions.

Gerstner, K. (1978). *Compendium for Literates*. Cambridge: MIT Press.

Harbison, S.P. and Steele, Jr., G.L. (1984). *C: A Reference Manual*. Prentice-Hall.

Kernighan, B.W. and Ritchie, D.M. (1978). *The C Programming Language*. Prentice-Hall.

Knuth, D.E. (1984). Literate Programming. *The Computer Journal* 27(2), 97-111.

Marcus, A. (1992). *Graphic Design for Electronic Documents and User Interfaces*. ACM Press.

Marcus, A. (1995). Principles of Effective Visual Communication for Graphical User Interface Design. In Baecker, R.M., Grudin, J., Buxton, W., and Greenberg, S. (1995). *Readings in Human Computer Interaction: Toward the Year 2000.* Morgan Kaufmann, 1995, 425-441.

Oman, P.W. (1997, to appear). *Programming Style Analysis*. Ablex.

Oman, P.W. and Cook, C.R. (1990a). The Book Paradigm for Improved Maintenance. *IEEE Software*, January 1990, 39-45.

Oman, P.W. and Cook, C.R. (1990b). Typographic Style is More than Cosmetic. *Communications of the ACM* 33(5), 506-520.

Ramsey, N. (1994). Literate Programming Simplified. *IEEE Software*, September 1994, 97-105.

Ruder, E. (1977). *Typographie*. Third Edition. Switzerland: Verlag Arthur Niggli Teufen AR.

Small, I.S. (1989). Program Visualization: Static Typographic Visualization in an Interactive Environment, M.Sc. Thesis, Department of Computer Science, University of Toronto, Feb. 1989.

Weizenbaum, J. (1969). Eliza — A Computer Program for the Study of Natural Language Communication between Man and Machine, *Communications of the ACM* 9(1), 36-45.